

Lesson 7: Model checking (II)

Model checking algorithms; SMV

G Stefanescu — National University of Singapore

Logic & Formal Systems
Semester 1, Fall, 2002-2003



Model checking algorithms

Recall that the problem to be solved by model checking is

(A) “Given a model \mathcal{M} , a CTL formula ϕ , and *a state* s , does $\mathcal{M}, s \models \phi$ hold or not?”, where

- \mathcal{M} is a model of the system and s is a state of the model;
- ϕ is a CTL formula intended to be satisfied by the system

What model for systems is using? As said (but with a more clear emphasis on the *finite* feature):

“A system is represented by a finite transition system (usually, a huge labeled directed graph, often with millions of states).”

Notice: The infinite trees obtained by unfolding such graphs are good to develop the intuition, but not to be used on our finite computers.



Model checking result

Given \mathcal{M}, s , and ϕ , the result returned by a model checker is either

(1) yes, $\mathcal{M}, s \models \phi$ or

(2) no $\mathcal{M}, s \not\models \phi$

but, quite useful, in the latter case most of the model checkers also return a trace/path which invalidates ϕ .

Alternative problem:

(B) Given a model \mathcal{M} and a CTL formula ϕ find *all states* s of the model which satisfy ϕ

Notice: The problems are obviously equivalent: once one is able to develop algorithms to solve one of them, the other is solved, as well. We will be mainly concerned with the *latter one*.



Reduced set of CTL connectives

We are starting with a version using the following reduced set of CTL connectives (see slides 6.23-24)

$$\Gamma = \{\perp, \neg, \wedge, \text{AF}, \text{EU}, \text{EX}\}$$

where:

- \perp , \neg , and \wedge are used for the propositional part
- AF, EU, and EX are used for the temporal part

Hence, there is a *preprocessing* procedure to:

1. check the CTL syntax correctness of the given formula ϕ and
2. translate it in a formula $\text{TRANSLATE}(\phi)$ written with connectives in Γ , only.

In the sequel, we suppose ϕ to be in CTL Γ -format.



The labeling algorithm

The idea of this algorithm is to:

1. decompose formula ϕ in pieces (sub-formulas) and apply a structural induction to label the graph with sub-formulas of ϕ (the intuition is that *a formula that labels a state is true in that state*)
2. for each such sub-formula, parse the graph to infer the truth in a state according to the meaning of the connectives and the truth values of its sub-formulas

In 2, one may need to know the values of sub-formulas in possibly many different states; this is the case for temporal operators, but not for the propositional ones.



..(the labeling algorithm)

Input: a CTL model $\mathcal{M} = (S, \rightarrow, L)$ and a CTL formula ϕ
(in Γ -format)

Output: the set of states of \mathcal{M} which satisfy ϕ

1. \perp : no states are labeled with \perp
2. p : label with p all states s such that $p \in L(s)$
3. $\neg\phi_1$: label s with $\neg\phi_1$ if s is not already labeled with ϕ_1
4. $\phi_1 \wedge \phi_2$: label s with $\phi_1 \wedge \phi_2$ if s is already labeled both with ϕ_1 and ϕ_2
5. $\text{EX } \phi_1$: label s with $\text{EX } \phi_1$ if one of its successors is already labeled with ϕ_1



...(the labeling algorithm)

6 AF ϕ_1 :

1. (initial marking) label any s with AF ϕ_1 if s is already labeled with ϕ_1
2. (repeated marking) label any s with AF ϕ_1 if all successor states of s are already labeled with AF ϕ_1
3. repeat (2) until there are no change

7 E[$\phi_1 \cup \phi_2$]:

1. (initial marking) label any s with E[$\phi_1 \cup \phi_2$] if s is already labeled with ϕ_2
2. (repeated marking) label any s with E[$\phi_1 \cup \phi_2$] if s is already labeled with ϕ_1 and at least one of its successor states is already labeled with E[$\phi_1 \cup \phi_2$]
3. repeat (2) until there are no change



Complexity

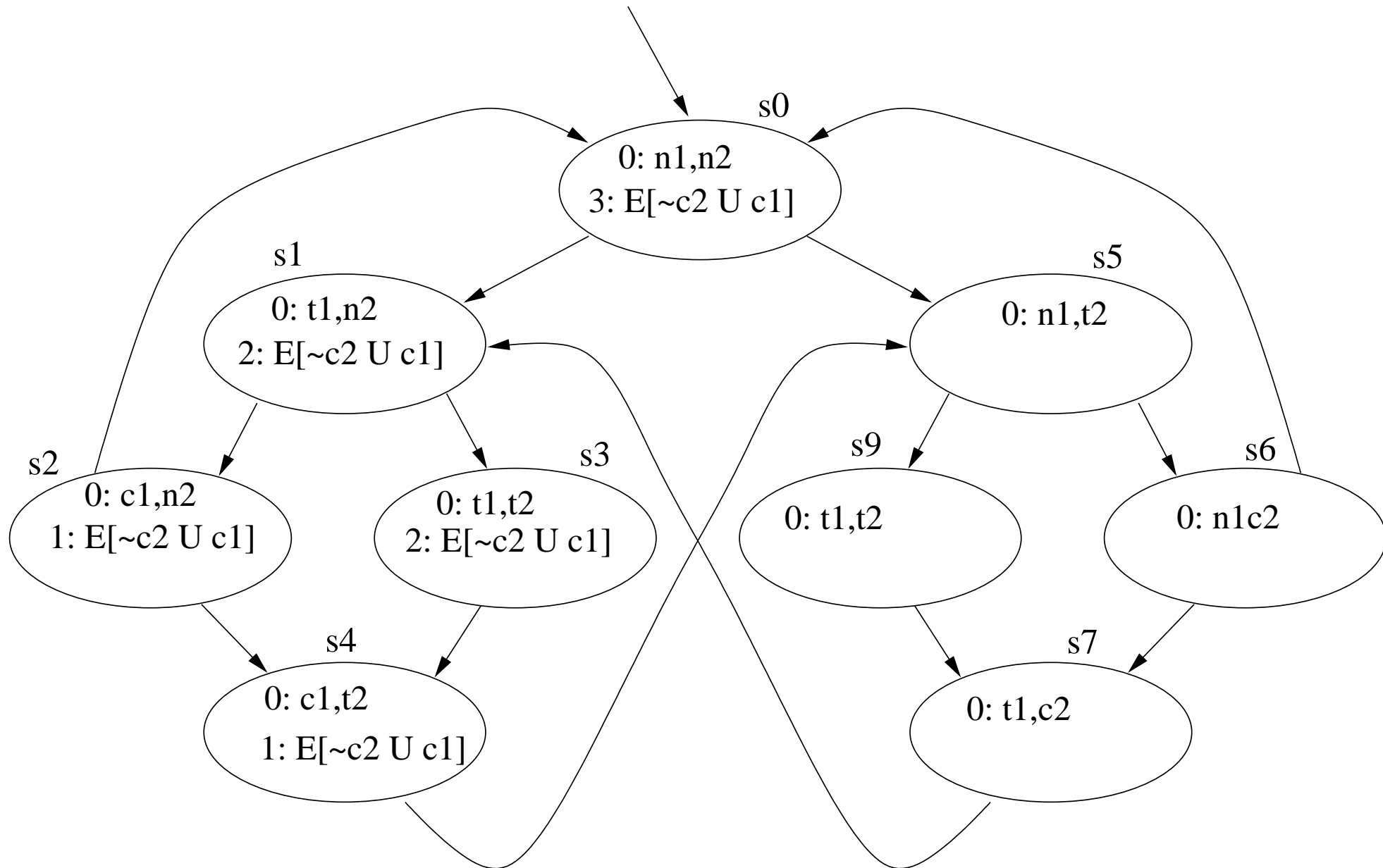
A rough analysis of the algorithm shows that it has the worse time complexity

$$O(k \cdot m \cdot (m + n))$$

where k is the number of connectives of the formula, m is the number of the states of the model, and n is the number of the transitions of the model.

Mutual exclusion, revisited

Checking $E[\neg c_2 \text{ U } c_1]$ in the second mutual exclusion model MUT2:





EG directly

EG may be handled directly as follows:

6' EG ϕ_1 :

0. label *all* states s with EG ϕ_1
1. (initial de-marking) if ϕ_1 does not hold in s then *delete* the label EG ϕ_1
2. (repeated de-marking) *delete* the label EG ϕ_1 from any state s if none of its successor states is labeled with EG ϕ_1
3. repeat (2) until there are no change

This different approach is based on the following greatest fixed-point characterization of EG

$$\text{EG } \phi \equiv \phi \wedge \text{EX EG } \phi$$



An improved variant

- Use EX, EU, and **EG** instead of EX, EU, and **AF**
- handle EX and EU as before (using backwards breadth-first searching)
- for EG ϕ
 - restrict to states satisfying ϕ
 - find SCCs (maximal strongly connected components; these are maximal regions such that any vertex is connected to any other vertex in the region)
 - use backwards breadth-first searching on the restricted graph to find any state that can reach an SCC

Complexity is reduced to $O(k \cdot (m + n))$ (k, m, n as before).



Pseudo-code

function SAT(ϕ):

/* precondition: ϕ is an arbitrary CTL formula */

/* postcondition: SAT(ϕ) returns the set of states satisfying ϕ */

begin function

case

ϕ is \top : **return** S

ϕ is \perp : **return** \emptyset

ϕ is atomic formula: **return** $\{s \in S : \phi \in L(s)\}$

ϕ is $\neg\phi_1$: **return** $S \setminus \text{SAT}(\phi_1)$

ϕ is $\phi_1 \wedge \phi_2$: **return** $\text{SAT}(\phi_1) \cap \text{SAT}(\phi_2)$

ϕ is $\phi_1 \vee \phi_2$: **return** $\text{SAT}(\phi_1) \cup \text{SAT}(\phi_2)$

ϕ is $\phi_1 \rightarrow \phi_2$: **return** $\text{SAT}(\neg\phi_1 \vee \phi_2)$

(...cont.)

...(pseudo-code)

(...cont.)

ϕ is $AX\phi_1$: **return** $SAT(\neg EX\neg\phi_1)$

ϕ is $EX\phi_1$: **return** $SAT_{EX}(\phi_1)$

ϕ is $A[\phi_1 U \phi_2]$: **return** $SAT(\neg(E[\neg\phi_1 U(\neg\phi_1 \wedge \neg\phi_2)] \vee EG\neg\phi_2))$

ϕ is $E[\phi_1 U \phi_2]$: **return** $SAT_{EU}(\phi_1, \phi_2)$

ϕ is $EF\phi_1$: **return** $SAT(E[\top U \phi_1])$

ϕ is $EG\phi_1$: **return** $SAT(\neg AF\neg\phi_1)$

ϕ is $AF\phi_1$: **return** $SAT_{AF}(\phi_1)$

ϕ is $AG\phi_1$: **return** $SAT(\neg EF\neg\phi_1)$

end case

end function



...(pseudo-code)

function $SAT_{EX}(\phi)$:

/* pre: ϕ is an arbitrary CTL formula */

/* post: $SAT_{EX}(\phi)$ returns the set of states satisfying $EX \phi$ */

local var X, Y

begin

$X := SAT(\phi)$;

$Y := \{s_0 \in S : s_0 \rightarrow s_1 \text{ for some } s_1 \in X\}$;

return Y

end



...(pseudo-code)

```
function SATAF( $\phi$ ):  
/* pre:  $\phi$  is an arbitrary CTL formula */  
/* post: SATAF( $\phi$ ) returns the set of states satisfying AF  $\phi$  */  
local var  $X, Y$   
begin  
   $X := S$ ;  
   $Y := \text{SAT}(\phi)$ ;  
  repeat until  $X = Y$   
    begin  
       $X := Y$ ;  
       $Y := Y \cup \{s \in S : \text{for all } s' \text{ with } s \rightarrow s' \text{ we have } s' \in Y\}$ ;  
    end  
  return  $Y$   
end
```



...(pseudo-code)

function $SAT_{EU}(\phi)$:

/ pre: ϕ is an arbitrary CTL formula */*

/ post: $SAT_{EU}(\phi, \psi)$ returns the set of states satisfying $E[\phi U \psi]$ */*

local var W, X, Y

begin

$W := SAT(\phi);$

$X := S;$

$Y := SAT(\psi);$

repeat until $X = Y$

begin

$X := Y;$

$Y := Y \cup (W \cap \{s \in S : \text{exists } s' \text{ such that } s \rightarrow s' \text{ and } s' \in Y\});$

end

return Y

end



The 'state explosion' problem

- the labeling algorithm is quite efficient [linear in the size of the model]
- ... but the model itself may be large, exponential in the number of the components (running in parallel 10 threads each of them having 10 states results in a systems with $10^{10} = 10,000,000,000$ states!)
- the tendency of the state space to become very large is commonly referred to as the *state explosion* problem
- the state explosion problem is mainly *unsolved* - no general solution is known at the moment



Dealing with ‘state explosion’

The problem is general unsolved. The following techniques were developed to overcome it in certain particular cases:

1. *efficient data structures* - e.g., *ordered binary decision diagrams OBDDs* (OBDDs are used to represent sets of states, not individual states)
2. *abstraction* - one may abstract away variables in the model that are not relevant for the formula being checked
3. *partial order reduction* - different runnings may be equivalent as far as the formula to be checked is concerned; partial order reduction check one trace from such a class only
4. *induction* - this technique is used when a large number of ‘identical’ processes is considered
5. *composition* - try to split the problem in small parts to be separately checked



SMV - Symbolic Model Verifier

SMV - Symbolic Model Verifier was one of the first model checkers. It is based on CTL, was developed in early '90, and had a strong impact on the verification field.

- SMV (Symbolic Model Verifier) was developing at CMU, see www.cs.cmu.edu/~modelcheck/smv.html
- it provides a language for describing the models/diagrams and it checks the validity of CTL formulas in such models
- the output is 'true' or a trace showing why the formula is false



SMV - Syntax

SMV - Syntax (informal)

- SMV programs consist of one or more modules (one of them should be `main`)
 - each module can declare variables and assign them
 - assignment uses two qualifications: `initial` (to indicate the initial states) and `next` (to indicate the next state in the corresponding state transition diagram)
 - the assignments may be nondeterministic - this is indicated by using the set notation `{...}` (choose one element from this set)
- (...cont.)



..(SMV - Syntax)

(...cont.)

- one may use the case construct; in such a case the conditions in front of ‘:’ are parsed from top to bottom and the first which is found true is executed; a default variant (with a always true condition, indicated by 1) is usually placed at the bottom of the case construct
- a module may have proper specifications to be checked, written in CTL syntax (but $\&$, $|$, $->$, $!$ are used instead of \wedge , \vee , \rightarrow , \neg)



SMV, 1st example

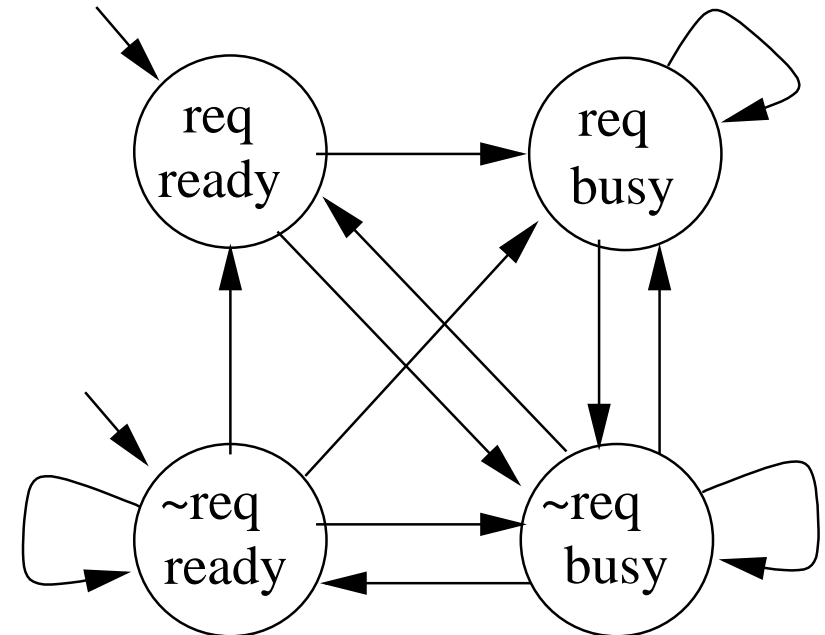
Our first program is rather typical:

- it models a part of the system which pass from ready to busy either due to some hidden reasons (not seen in the model) or due to a visible request request;
- the system pass from busy to ready in a nondeterministic way, too (no visible reason)
- the intention of this simple abstract model is to check if it satisfies the formula

$AG(\text{request} \rightarrow AF \text{ status} = \text{busy})$

...(SMV, 1st example)

```
MODULE main
VAR
  request : boolean;
  status : {ready,busy};
ASSIGN
  init(status) := ready;
  next(status) :=
    case
      request : busy;
      1 : {ready,busy};
    esac;
SPEC
  AG(request -> AF status = busy)
```





SMV, 2nd example

The 2nd program illustrates the use of modules:

- the program models a counter from 000 to 111
- a module `counter_cell` is instantiated 3 times with names `bit0`, `bit1`, and `bit2`
- `counter_cell` has a formal parameter
- the period ‘.’ is used to access the variables of a particular instance (`m.v` indicates a reference to the variable `v` of module `m`)
- we check the following easy formula
`AG AF bit2.carry_out`



...(SMV, 2st example)

```
MODULE main
VAR
    bit0 : counter_cell(1);
    bit1 : counter_cell(bit0.carry_out);
    bit2 : counter_cell(bit1.carry_out);
SPEC
    AG AF bit2.carry_out

MODULE counter_cell(carry_in)
VAR
    value : boolean;
ASSIGN
    init(value) := 0;
    next(value) := value + carry_in mod 2;
DEFINE
    carry_out := value & carry_in;
```



SMV, 2nd example

Notice: define statement is used to avoid increasing the state space; its effect may be obtained with a variable, too:

```
VAR
```

```
    carry_out : boolean;
```

```
ASSIGN
```

```
    carry_out := value & carry_in;
```



Synchronous and asynchronous composition

By default, SMV modules are composed *synchronously*:

at each clock tick, each module executes a transition

(mainly used for hardware verification)

It is also possible to model *asynchronous* composition

at each clock tick, SMV chooses a module in a random way
and executes a transition there

(mainly used for verifying communication protocols)



SMV, 3rd example - Mutual Exclusion

A CTL model for ‘mutual exclusion problem’ was presented before. Here we give a SMV implementation. A few new features are:

- there is a module `main` with (1) a variable `turn` which determines the process to enter in its critical section and (2) two instantiations of the module `prc`
- because of the `turn` variable the state transition diagram (shown later) is slightly more complicated
- one important new feature is the presence of the `fairness` statement; it contains a CTL formula ϕ and restricts the search to those paths where ϕ is true infinitely often (running is an SMV keyword indicating that the corresponding module is selected for execution infinitely often)



...(SMV, 3rd example)

```
MODULE main
  VAR
    pr1 : process prc(pr2.st, turn, 0);
    pr2 : process prc(pr1.st, turn, 1);
    turn : boolean;
  ASSIGN
    init(turn) := 0;
  --safety
  SPEC AG!((pr1.st = c) & (pr2.st = c))
  --liveness
  SPEC AG((pr1.st = t) -> AF (pr1.st = c))
  SPEC AG((pr2.st = t) -> AF (pr2.st = c))
  --no strict sequencing
  SPEC EF(pr1.st = c & E[pr1.st = c U
    (!pr1.st = c & E[! pr2.st = c U pr1.st = c ])])
```

...(SMV, 3rd example)

```
MODULE prc(other-st, turn, myturn)
  VAR
    st : {n, t, c};
  ASSIGN
    init(st) := n;
    next(st) :=
      case
        (st = n) : {t, n};
        (st = t) & (other-st = n) : c;
        (st = t) & (other-st = t) & (turn = myturn) : c;
        (st = c) : {c, n};
      1 : st;
      esac;
    next(turn) :=
      case
        turn = myturn & st = c : !turn;
      1 : turn;
      esac;
  FAIRNESS running
  FAIRNESS !(st = c)
```

...(SMV, 3rd example)

Mutual exclusion in SMV:

